UML and MARTE profile

Emad Ebeid
PhD Student
Department of Computer Science
University of Verona
Italy

Davide Quaglia
Assistant Professor
Department of Computer Science
University of Verona
Italy

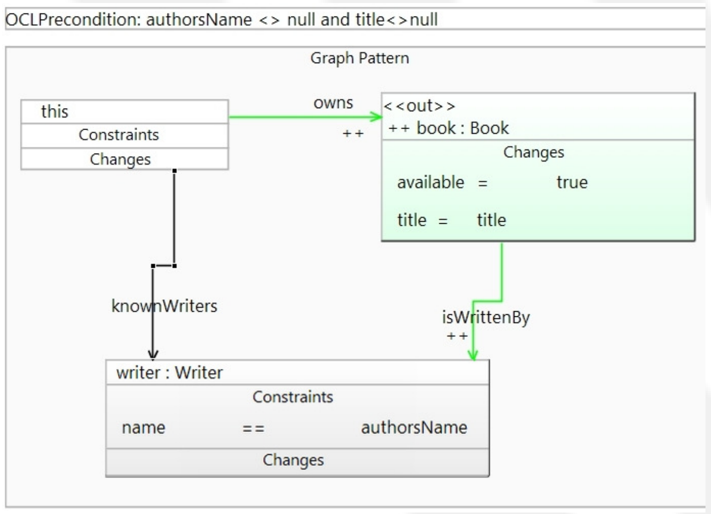
Overview

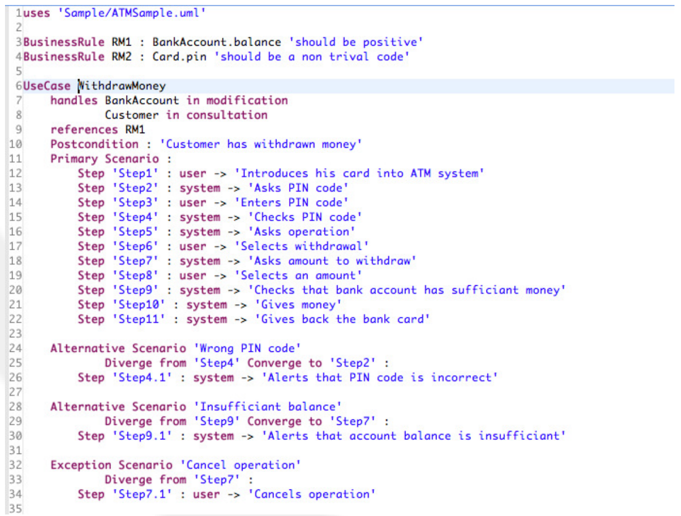
- What is Modeling language?
- What is UML?
- A brief history of UML
- Understanding the basics of UML
- UML diagrams
- UML Profiles
- MARTE profile
- UML Modeling tools

What is Modeling language?

A modeling language is any artificial language that can be used to express *information*, *knowledge* or *systems* in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure

- A modeling language can be graphical or textual

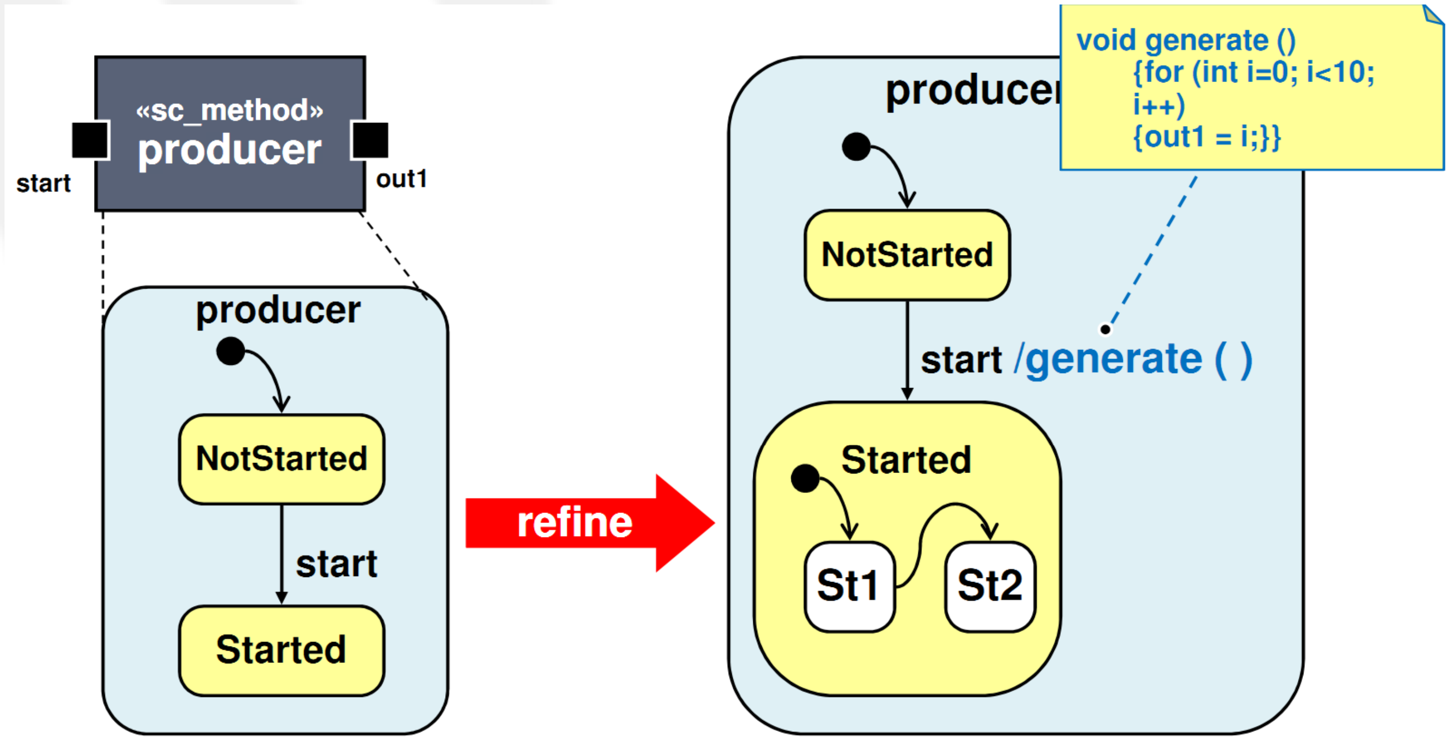




```
uses 'Sample/ATMSample.uml'

BusinessRule RM1 : BankAccount.balance 'should be positive'
BusinessRule RM2 : Card.pin 'should be a non trival code'

UseCase WithdrawMoney
    handles BankAccount in modification
            Customer in consultation
    references RM1
    Postcondition : 'Customer has withdrawn money'
    Primary Scenario :
        Step 'Step1' : user -> 'Introduces his card into ATM system'
        Step 'Step2' : system -> 'Asks PIN code'
        Step 'Step3' : user -> 'Enters PIN code'
        Step 'Step4' : system -> 'Checks PIN code'
        Step 'Step5' : system -> 'Asks operation'
        Step 'Step6' : user -> 'Selects withdrawal'
        Step 'Step7' : system -> 'Asks amount to withdraw'
        Step 'Step8' : user -> 'Selects an amount'
        Step 'Step9' : system -> 'Checks that bank account has sufficiant money'
        Step 'Step10' : system -> 'Gives money'
        Step 'Step11' : system -> 'Gives back the bank card'

    Alternative Scenario 'Wrong PIN code'
            Diverge from 'Step4' Converge to 'Step2' :
        Step 'Step4.1' : system -> 'Alerts that PIN code is incorrect'

    Alternative Scenario 'Insufficiant balance'
            Diverge from 'Step9' Converge to 'Step7' :
        Step 'Step9.1' : system -> 'Alerts that account balance is insufficiant'

    Exception Scenario 'Cancel operation'
            Diverge from 'Step7' :
        Step 'Step7.1' : user -> 'Cancels operation'

```

Model-based development

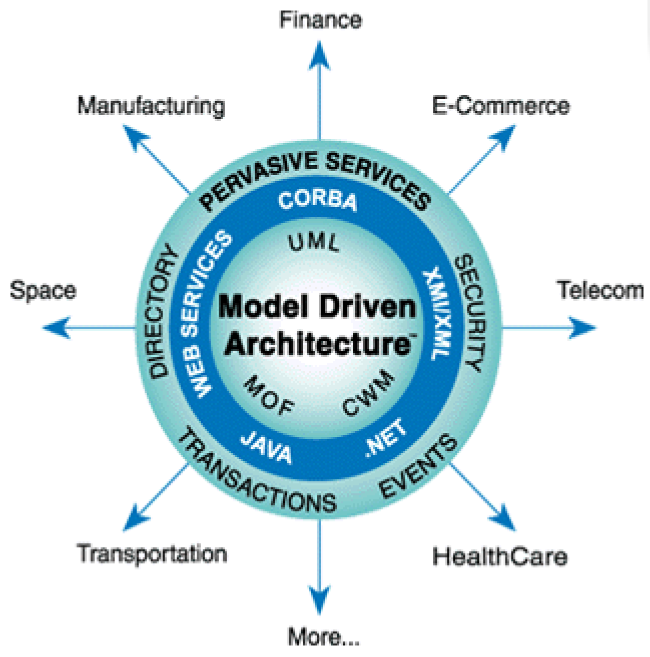
- Models can be refined continuously until the application is fully specified



```
void generate ()
     {for (int i=0; i<10;
     i++)
     {out1 = i;}}
```

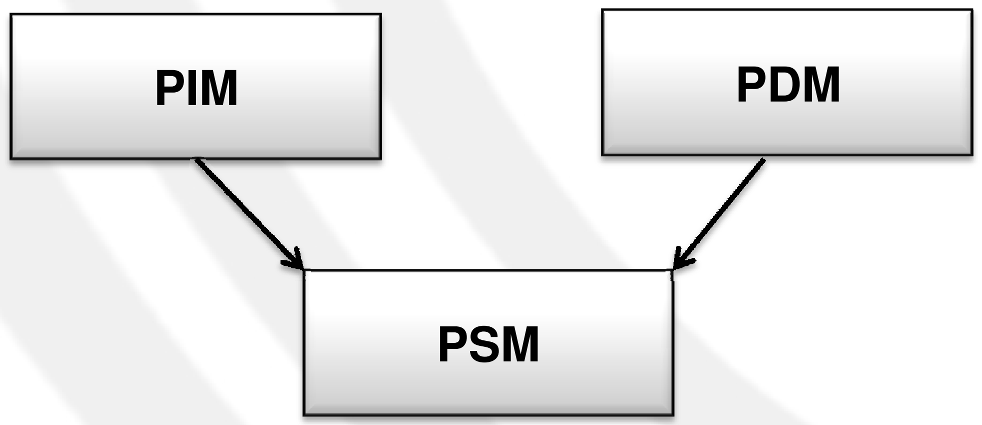
Model-Driven Architecture (MDA)™

- It was launched by the Object Management Group (OMG) in 2001
- MDA provide portability, interoperability, maintainability and reusability of models
- MDA approach defines system functionality using a platform-independent model (PIM) using an **appropriate domain-specific language**



Model-driven architecture viewpoints

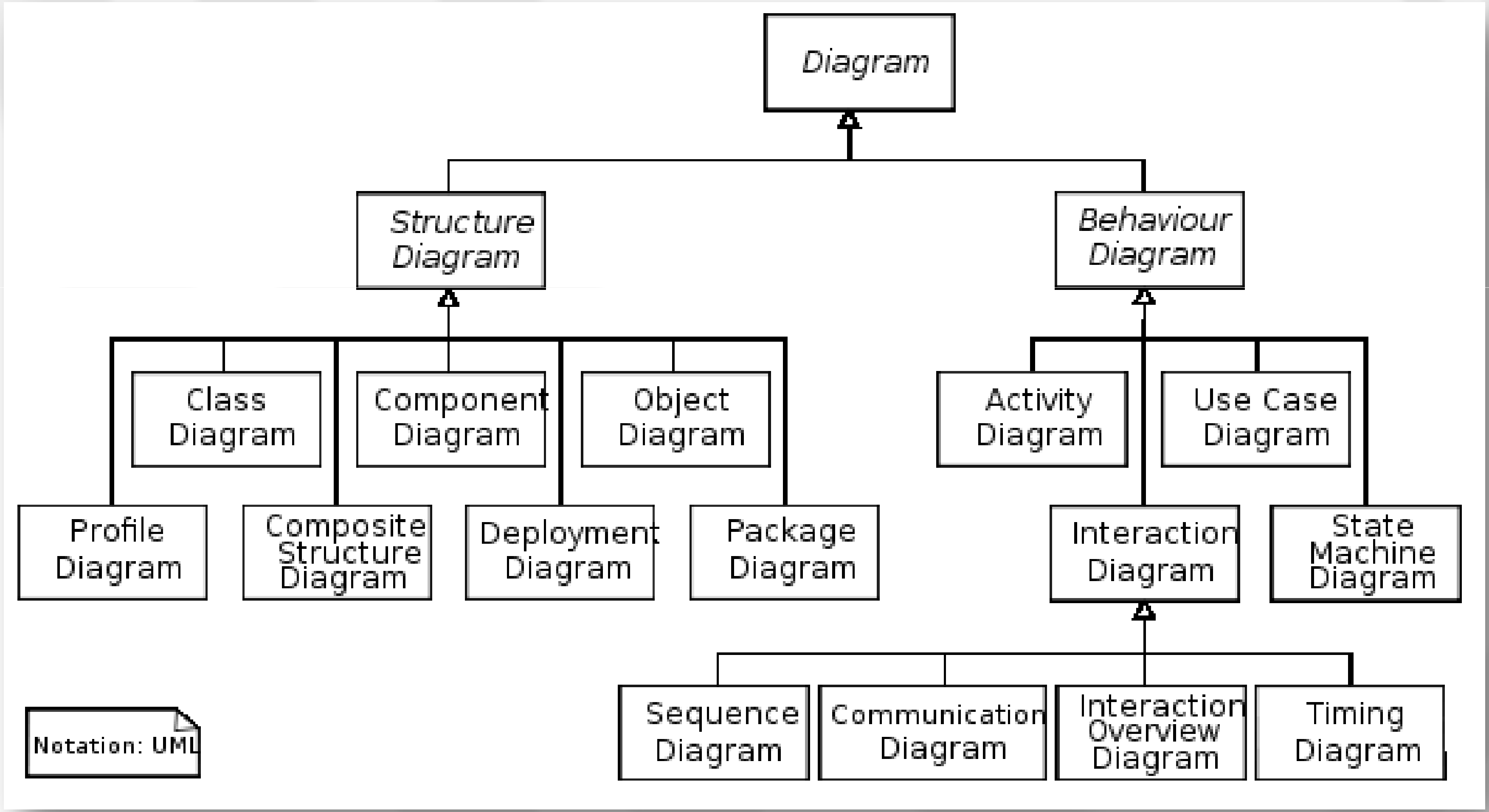
- **The Platform Independent Model (PIM):** The functional and non-functional aspects
- **The Platform Description Model (PDM):** HW and SW resources
- **The Platform Specific Model (PSM):** System architecture



What is UML?

- **Unified Modeling Language (UML)** is a standardized general-purpose modeling language in the field of object-oriented software engineering
- The standard was created, and is managed by the Object Management Group

UML diagrams



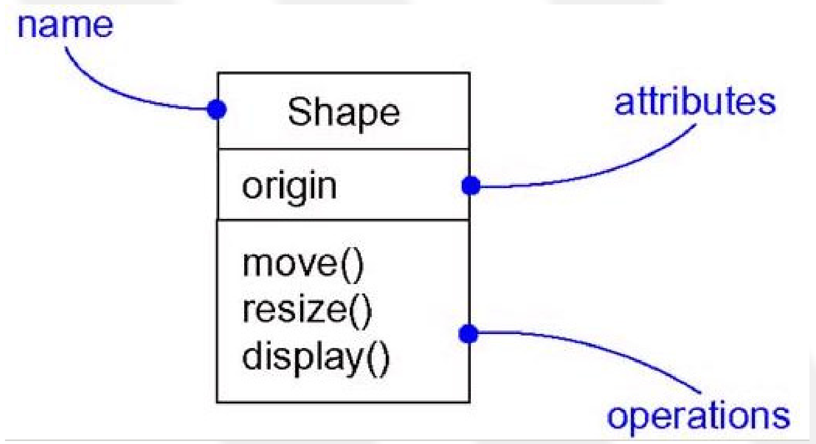
Why UML for Modeling

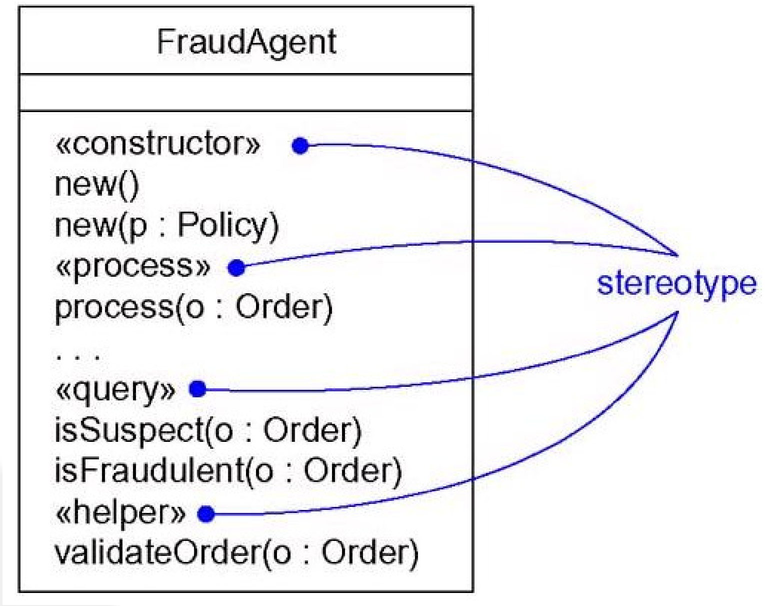
- Use graphical notation to communicate more clearly than natural language (imprecise) and code(too detailed).
- Help acquire an overall view of a system.
- UML is *not* dependent on any one language or technology.
- UML moves us from fragmentation to standardization.

Class Diagram

- Modeling a system involves identifying the things that are important to your particular view.
- In the UML, all of these things are modeled as classes.
- A class is an abstraction of the things that are a part of your vocabulary.
- A class is not an individual object, but rather represents a whole set of objects.
- An *attribute* is a named property of a class that describes a range of values that instances of the property may hold.
- An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior.
- To better organize long lists of attributes and operations, you can also prefix each group with a descriptive category by using stereotypes.

Class Diagram (2)

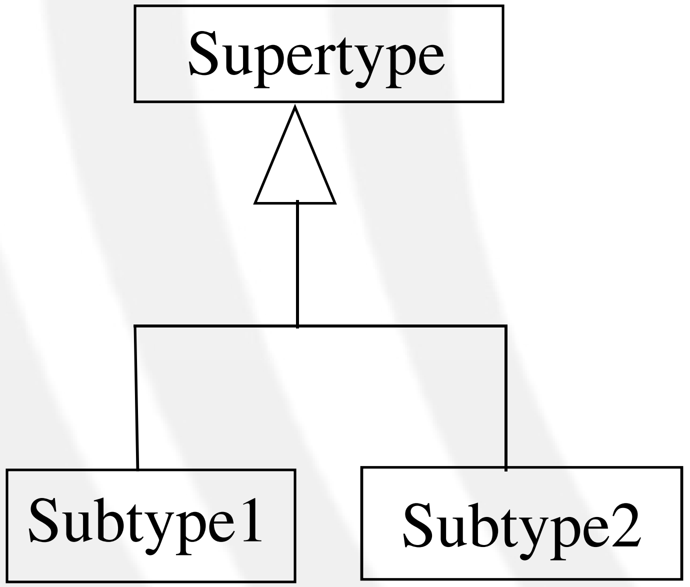




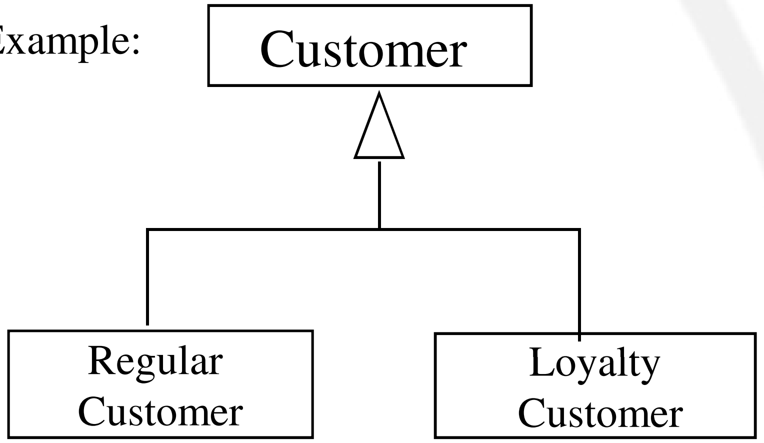
OO Relationships

- There are two kinds of Relationships
  - Generalization (parent-child relationship)
  - Association (student enrolls in course)
- Associations can be further classified as
  - Aggregation
  - Composition

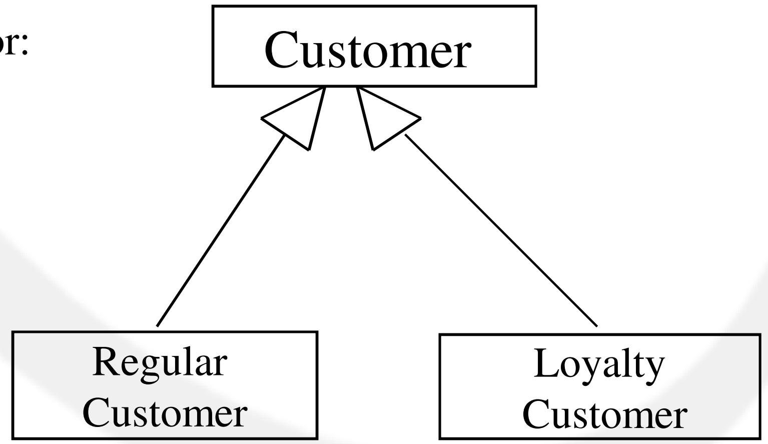
OO Relationships: Generalization



Example:



or:



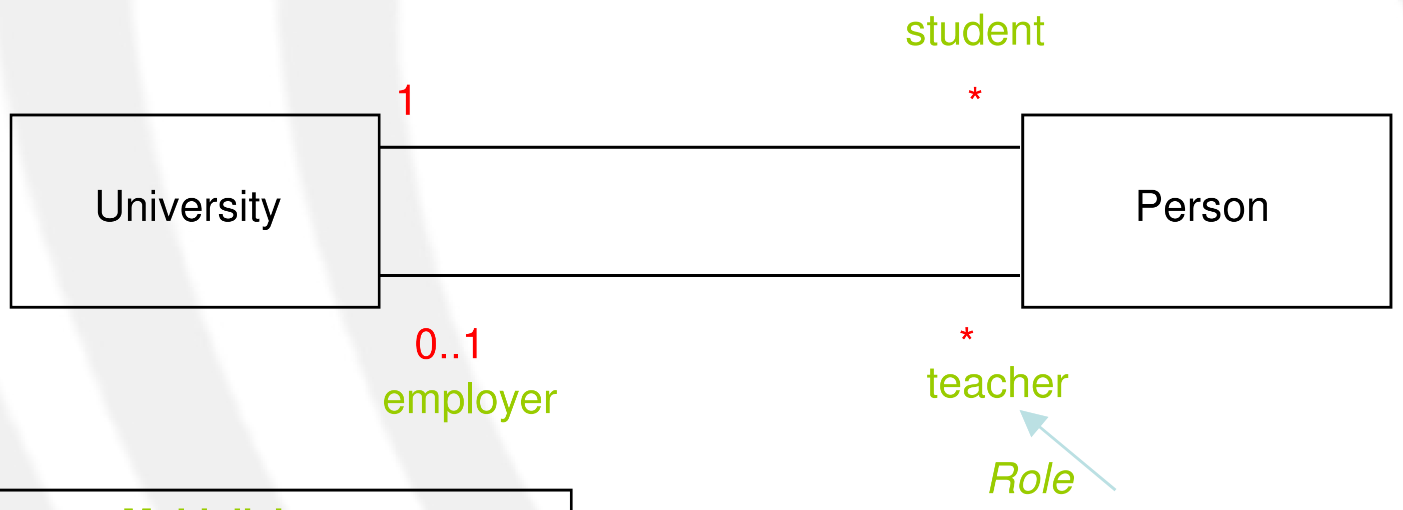
- Generalization expresses a parent/child relationship among related classes.

- Used for abstracting details in several layers

OO Relationships: Association

- Represent relationship between instances of classes
  - Student enrolls in a course
  - Courses have students
  - Courses have exams
  - Etc.
- Association has two ends
  - Role names (e.g. enrolls)
  - Multiplicity (e.g. One course can have many students)

Association: Multiplicity and Roles



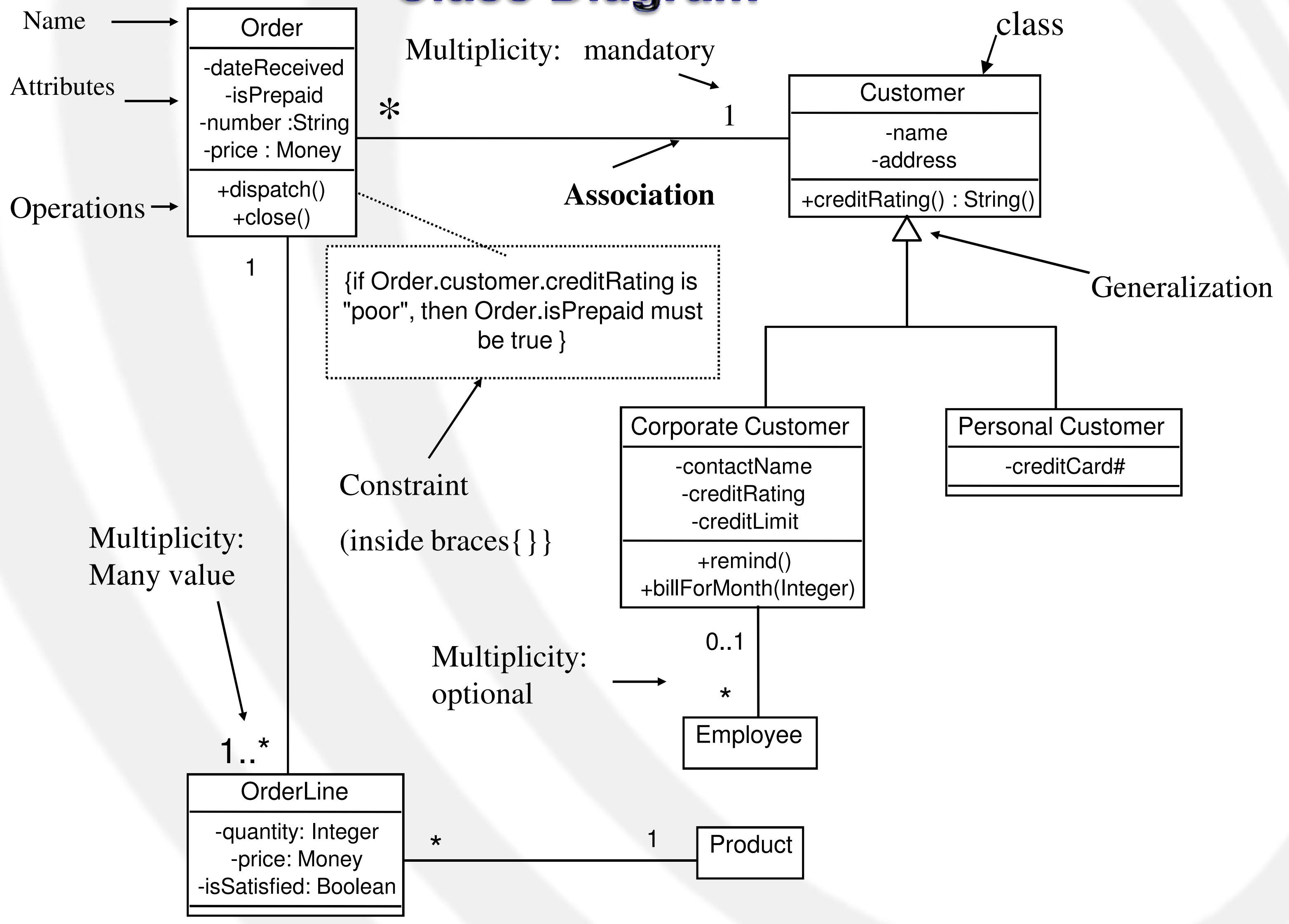
Multiplicity

Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

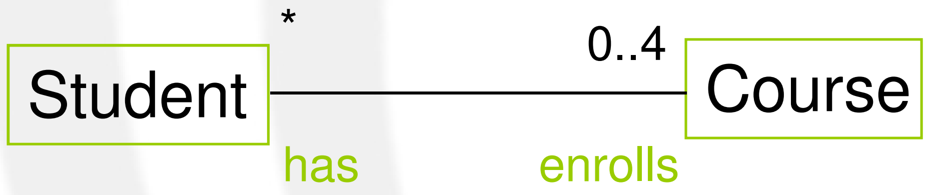
Role

"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

Class Diagram

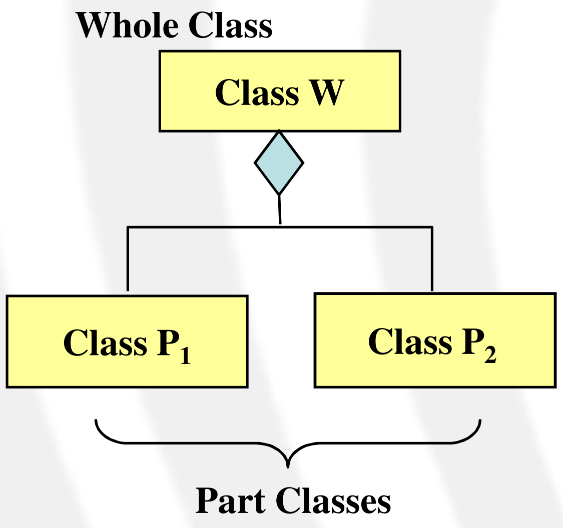


Association: Model to Implementation

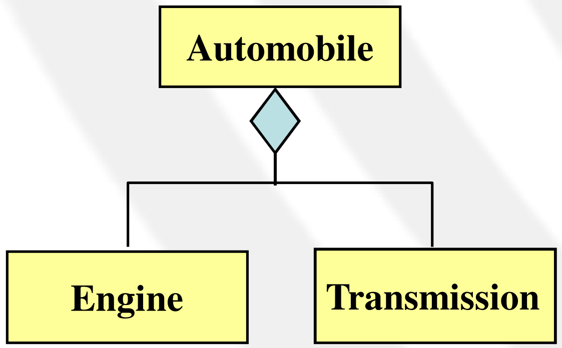


```
Class Student {
    Course enrolls[4];
}

Class Course {
    Student have[];
}
```

OO Relationships: Composition



Example



Composition: expresses a relationship among instances of related classes. It is a specific kind of Whole-Part relationship.

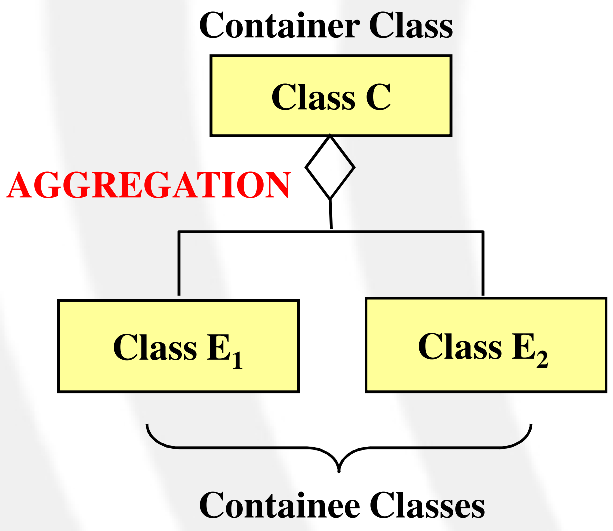
It expresses a relationship where an instance of the Whole-class has the responsibility to create and initialize instances of each Part-class.

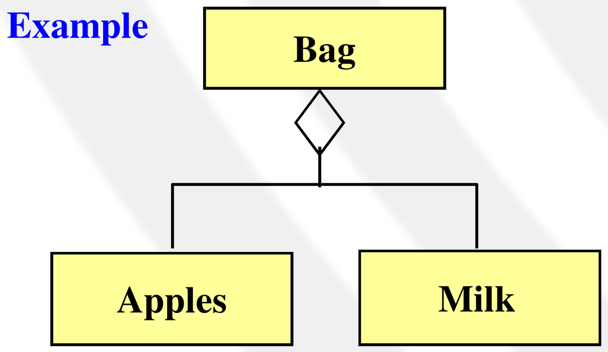
It may also be used to express a relationship where instances of the Part-classes have privileged access or visibility to certain attributes and/or behaviors defined by the Whole-class.

Composition should also be used to express relationship where instances of the Whole-class have exclusive access to and control of instances of the Part-classes.

Composition should be used to express a relationship where the behavior of Part instances is undefined without being related to an instance of the Whole. And, conversely, the behavior of the Whole is ill-defined or incomplete if one or more of the Part instances are undefined.

OO Relationships: Aggregation





Aggregation: expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

It expresses a relationship where an instance of the Container-class has the responsibility to hold and maintain instances of each Containee-class that have been created outside the auspices of the Container-class.

Aggregation should be used to express a more informal relationship than composition expresses. That is, it is an appropriate relationship where the Container and its Containees

Aggregation is appropriate when Container and Containees have no special access privileges to each other.

Aggregation vs. Composition

•**Composition** is really a strong form of **aggregation**

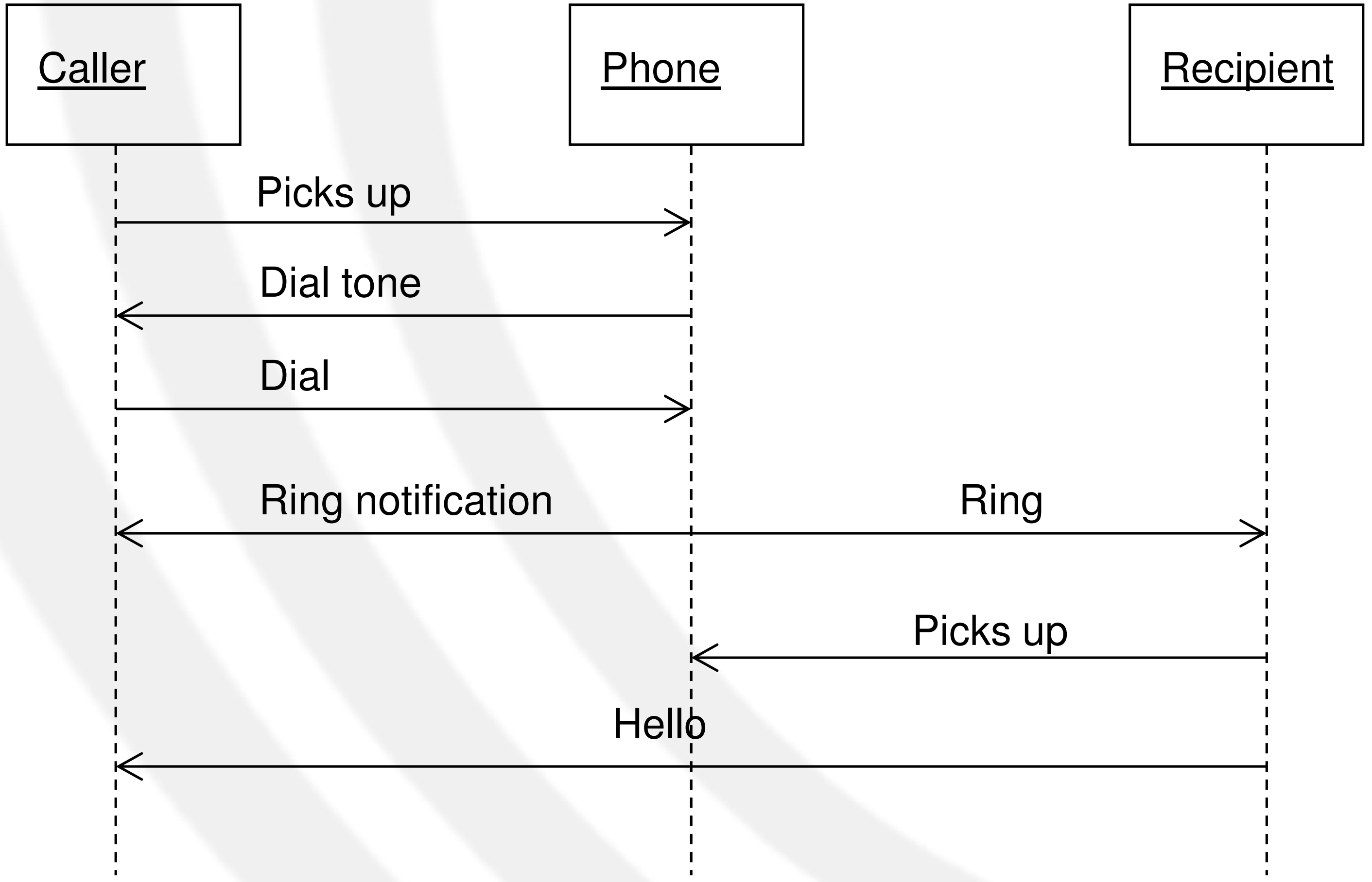
- •components have only one owner
- •components cannot exist independent of their owner
- •components live or die with their owner

e.g. Each car has an engine that can not be shared with other cars.

•**Aggregations** may form "part of" the aggregate, but may not be essential to it. They may also exist independent of the aggregate.

e.g. Apples may exist independent of the bag.

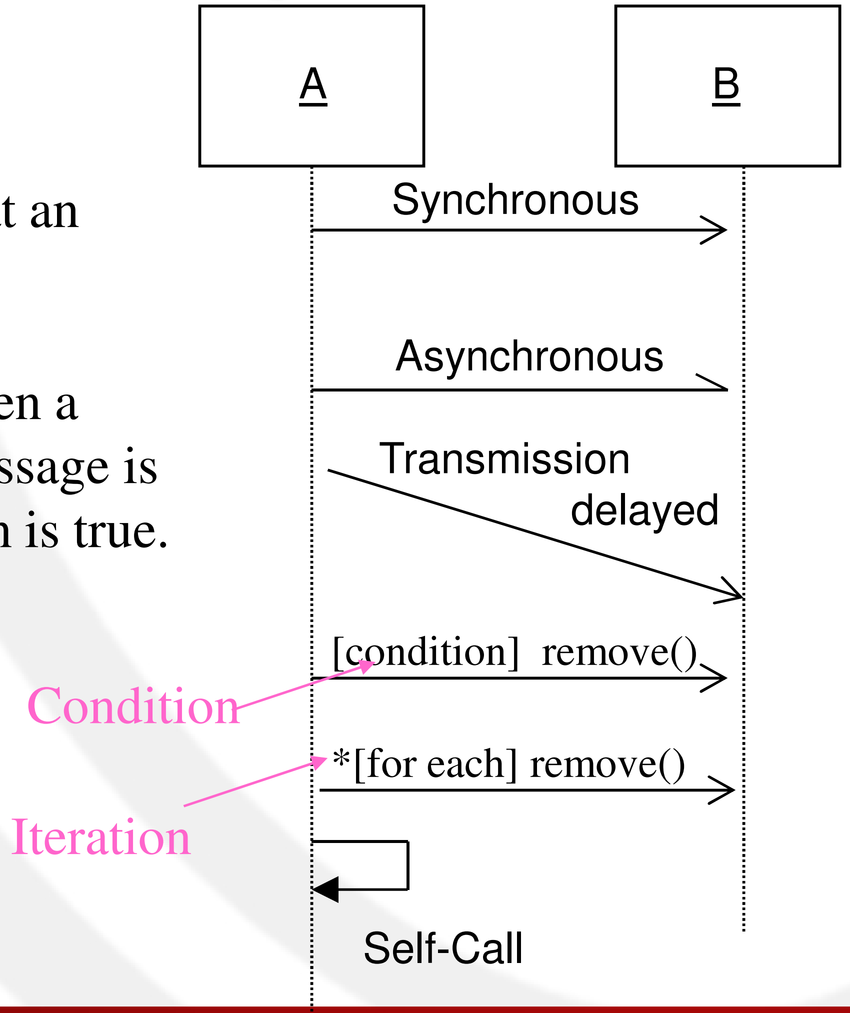
Sequence Diagram(make a phone call)



Sequence Diagram:Object interaction

Self-Call: A message that an Object sends to itself.

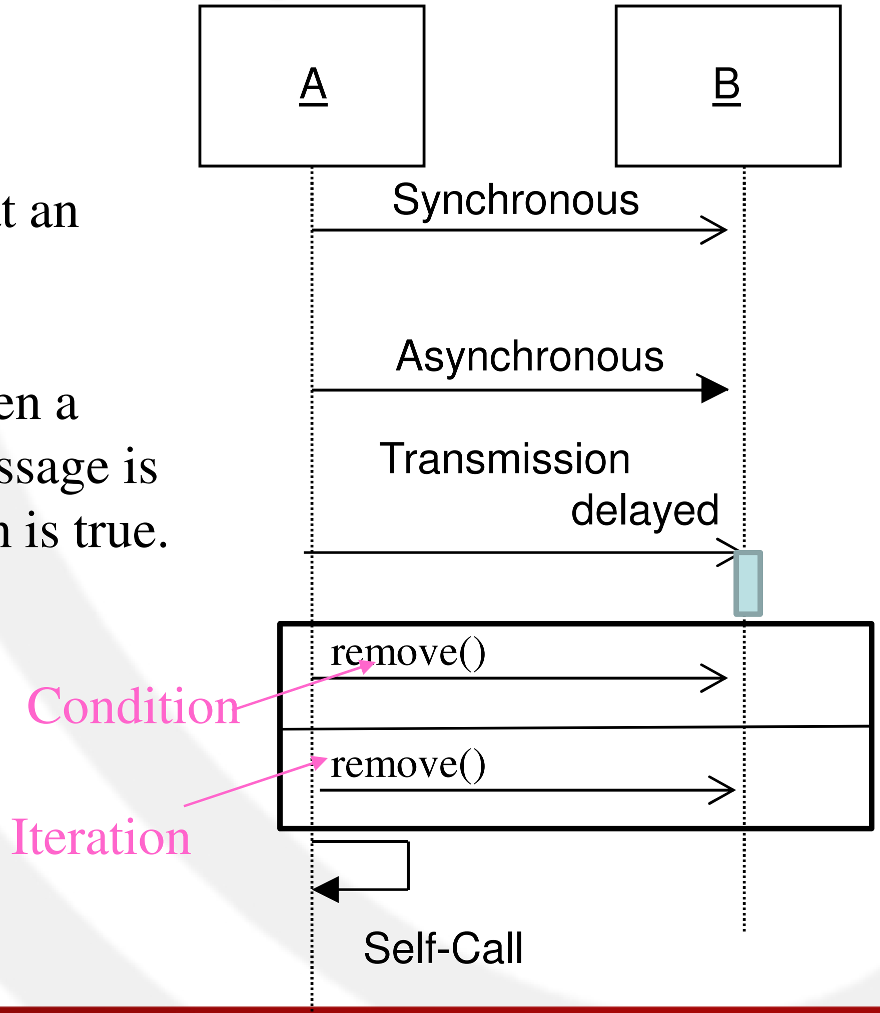
Condition: indicates when a message is sent. The message is sent only if the condition is true.



Sequence Diagram:Object interaction

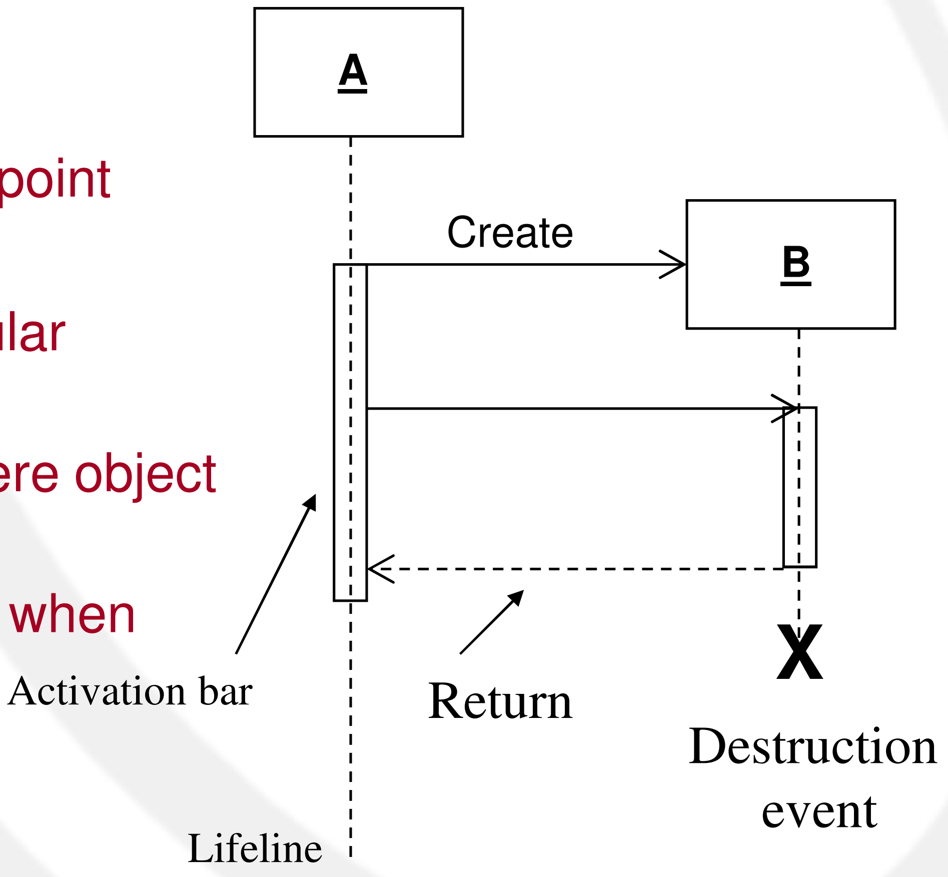
Self-Call: A message that an Object sends to itself.

Condition: indicates when a message is sent. The message is sent only if the condition is true.



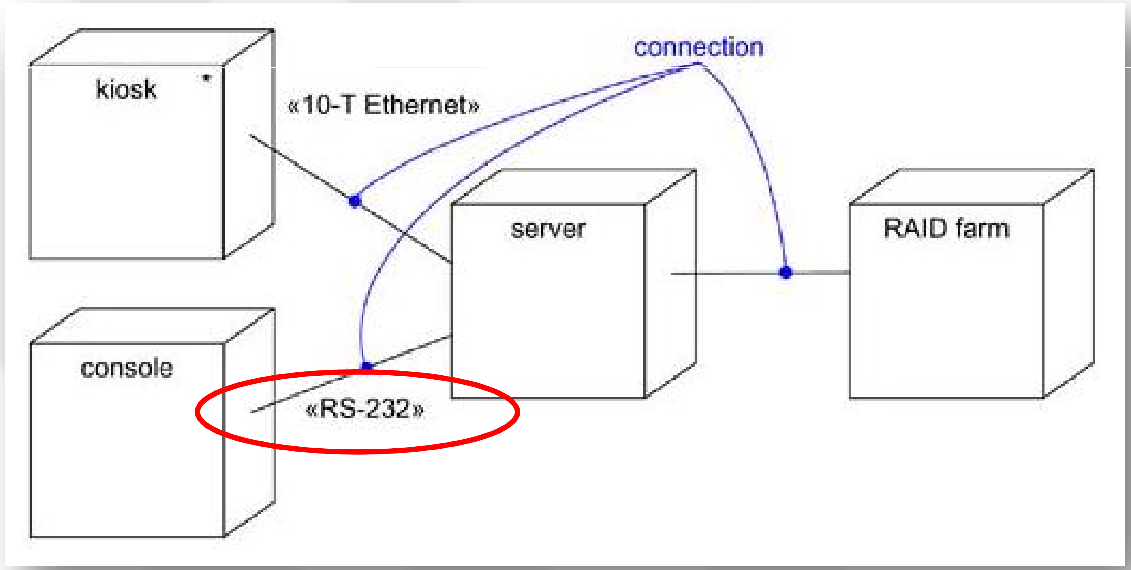
Sequence Diagrams – Object Life Spans

- Creation
  - Create message
  - Object life starts at that point
- Activation
  - Symbolized by rectangular stripes
  - Place on the lifeline where object is activated.
  - Rectangle also denotes when object is deactivated.
- Destruction event
  - Placing an ‘X’ on lifeline
  - Object’s life ends at that point



Deployment Diagram

- The components must be deployed on <u>some set of hardware</u> in order to execute.



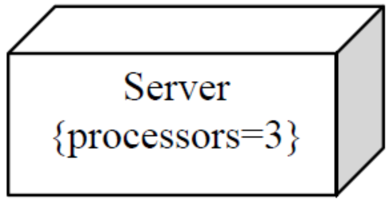
UML Profiles

- **Profile**: Provides a generic extension mechanism for customizing UML models for particular domains and platforms. Extension mechanisms allow refining standard semantics in strictly additive manner

- Profiles are defined using stereotypes, tag definitions, and constraints that are applied to specific model elements, such as Classes, Attributes, Operations, and Activities

- A Profile is a collection of such extensions that collectively customize UML for a particular domain (e.g., aerospace, healthcare, financial) or platform (J2EE, .NET)

Tagged Values



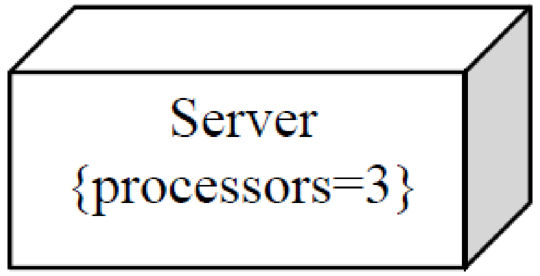
A tagged value is a combination of a tag and a value that gives supplementary information that is attached to a model element. A tagged value can be used to add properties to any model elements and can be applied to a model element or a stereotype.

Tagged values can be defined for existing model elements, or for individual stereotypes, so that everything with that stereotype has that tagged value. It is important to mention that a tagged value is not equal to a class attribute. Instead, you can regard a tagged value as being a metadata, since its value applies to the element itself and not to its instances.

One of the most common uses of a tagged value is to *specify properties* that are relevant to code generation or configuration management. So, for example, you can make use of a tagged value in order to specify the programming language to which you map a particular class, or you can use it to denote the author and the version of a component.

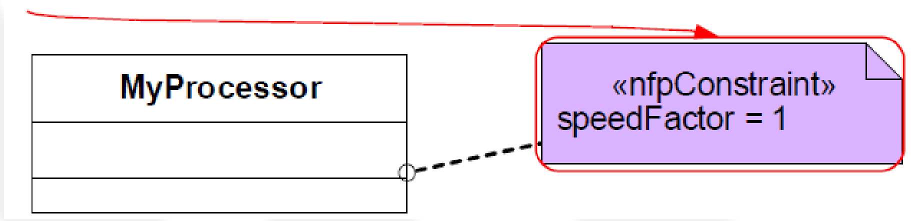
Tagged Values

- Graphically, a tagged value is rendered as a string enclosed by brackets, which is placed below the name of another model element. The string consists of a name (the tag), a separator (the symbol =), and a value (of the tag)



Constraints

- Constraints are properties for specifying semantics and/or conditions that must be held true at all times for the elements of a model. They allow you to extend the semantics of a UML building block by adding new rules, or modifying existing ones.
- For example, when modeling hard real time systems it could be useful to adorn the models with some additional information, such as time budgets and deadlines. By making use of constraints these timing requirements can easily be captured.



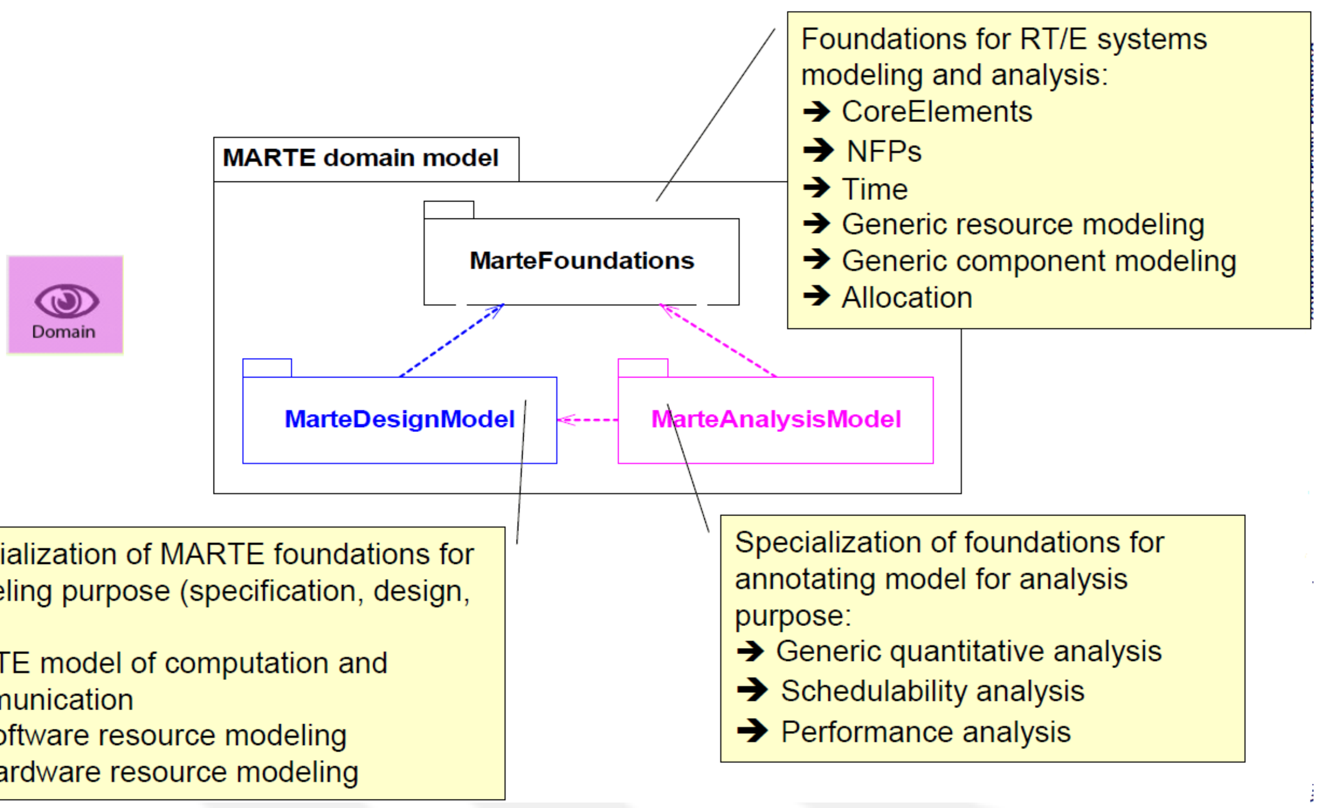
Catalog of Adopted OMG Profiles

- UML Profile for CORBA
- UML Profile for Enterprise Application Integration (EAI)
- UML Profile for Enterprise Distributed Object Computing (EDOC)
- UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms
- UML Profile for Schedulability, Performance, and Time
- UML Profile for System on a Chip (SoC)
- UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE)
- UML Testing Profile
- UML Profile for Systems Engineering (SysML)
- UML Profile for DoDAF/MoDAF (UPDM)

MARTE profile

- ***MARTE*** (Modelling and Analysis Real-Time and Embedded systems) deals with time- and resource-constrained aspects, and includes a detailed taxonomy of hardware and software patterns along with their non-functional attributes to enable state-of-the art quantitative analyses (e.g., performance and power consumption)

MARTE overview



Foundations for RT/E systems modeling and analysis:
➔ CoreElements
➔ NFPs
➔ Time
➔ Generic resource modeling
➔ Generic component modeling
➔ Allocation

Specialization of MARTE foundations for modeling purpose (specification, design, …):
➔ RTE model of computation and communication
➔ Software resource modeling
➔ Hardware resource modeling

Specialization of foundations for annotating model for analysis purpose:
➔ Generic quantitative analysis
➔ Schedulability analysis
➔ Performance analysis

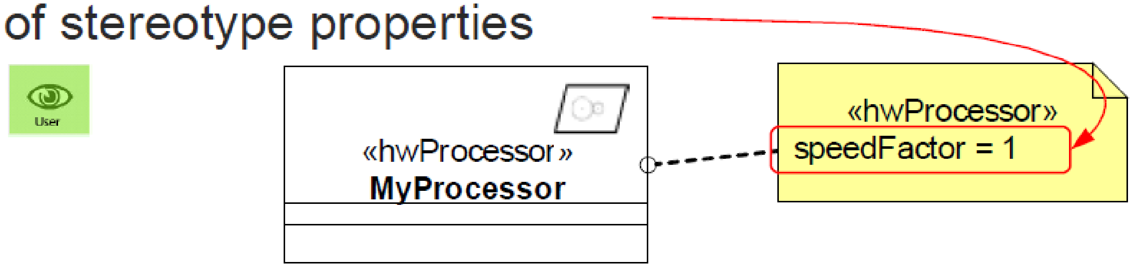
Non-Functional Properties (NFPs)

- Non-functional properties describe the “fitness” of systems behavior. (E.g., performance, memory usage, power consumption,..etc)

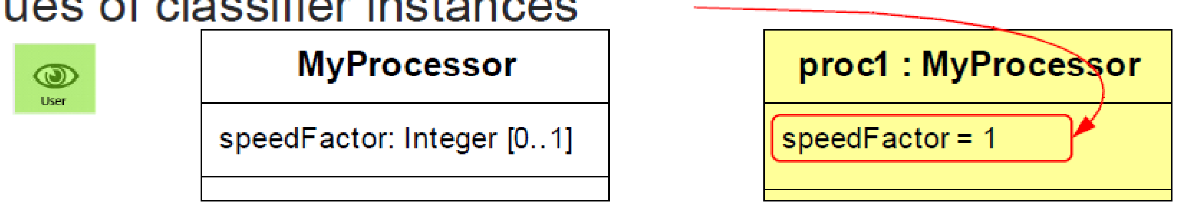
NFP subprofile

Three mechanisms to annotate UML models:

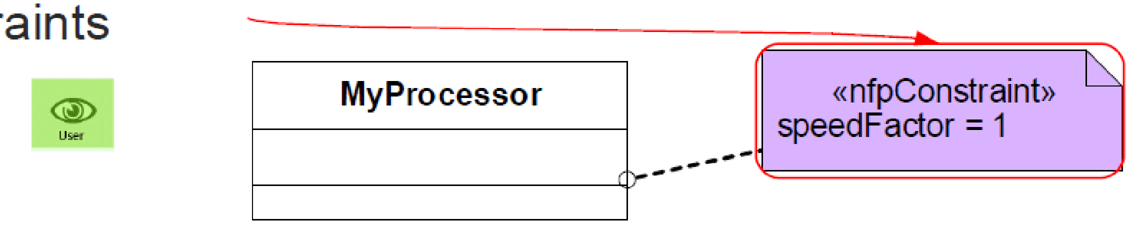
- Values of stereotype properties



- Slot values of classifier instances

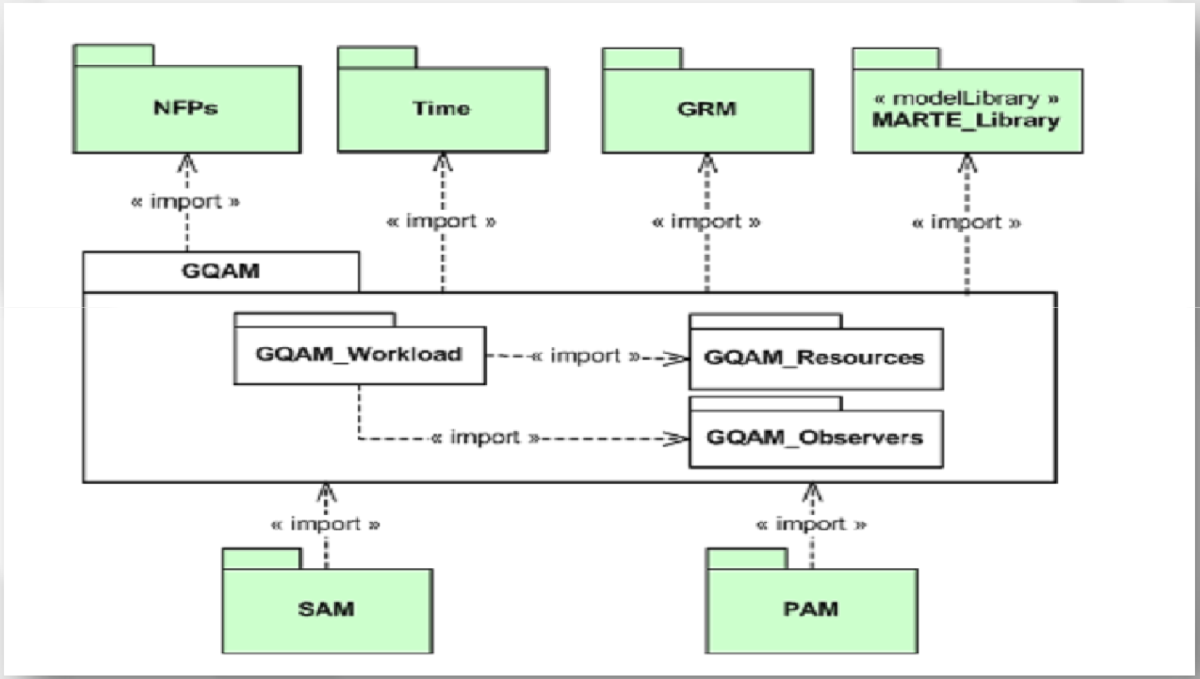


- Constraints



Generic Quantitative Analysis Modeling (GQAM)

The generic analysis domain includes specialized domains in which the analysis is based on the software behavior, such as performance and schedulability and also power, memory, reliability, availability, and security.



GaExecHost

- It denotes a processor that executes Steps
- In performance modeling, an GaExecHost can be any device which executes behavior, including storage and peripheral devices.

node

UML Modeling Tools

- Rational Rose (www.rational.com) by IBM
- TogetherSoft Control Center, Borland (http://www.borland.com/together/index.html)
- ArgoUML (free software) (http://argouml.tigris.org/)
  OpenSource; written in java
- Others (http://www.objectsbydesign.com/tools/umltools_byCompany.html)

Reference

1. **UML Distilled:** A Brief Guide to the Standard Object Modeling Language
   Martin Fowler, Kendall Scott

2. IBM Rational
   http://www-306.ibm.com/software/rational/uml/

3. Practical UML --- A Hands-On Introduction for Developers
   http://www.togethersoft.com/services/practical_guides/umlonlinecourse/

4. Software Engineering Principles and Practice. Second Edition;
   Hans van Vliet.

5. http://www-inst.eecs.berkeley.edu/~cs169/

Any questions?